Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Genericity

Philippe Collet

Master 1 IFI International
2012-2013
http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1213

Agenda

- Generics in Java 5 (studied in labs)
- Introduction
- Principles of parameterization
- Principles of genericity
- Genericity in OO languages
- Static and dynamic genericity

Université Nice SOPHIA ANTIPOLIS

On software evolution

- Mastering evolution is one of the major challenges of software engineering: *maintenance cost, qualities deterioration, oversight, bugs…*
- **Mastering evolution** consists in:
  - Introducing needed modifications
  - Only needed ones
  - Wherever this is needed
  - Without impairing initial qualities
  - At the lower cost…

Difficult Problem

especially if the software is large and complex

On software evolution (cont'd)

- A lot of software techniques aim at facilitating evolution management:
  - Avoid duplication (uniqueness in definition)
  - Hide what is not useful (encapsulation)
  - Locate and explain dependencies
- Two families of techniques to manage evolution:
  - **Planned techniques**
    - Anticipate possible evolutions, to make them easier and safer
  - **Adaptive techniques**
    - « lazy » change handling, one at a time, but in the most automatic way

Planned vs. Adaptive

- Planned
  - Parameterization, factorization, genericity, formal models, model-driven engineering
  - Raising the level of abstraction, overhead of creation should be compensated by ROI
- Adaptive
  - Rewriting, inheritance, separation of concerns, dynamically reconfigurable components

- Each technique borrows a small part of characteristics from the other:
  - There is some planning in inheritance
  - There is some adaptiveness in genericity
- Both techniques complement themselves and can combine with eachother

Principles of parameterization

- Define a formal parameter of an entity, abstract and general, which displays what is necessary and sufficient
- Substitute in an automatic way formal by effective parameters, with no impact on the use of the formal ones

- Verify that:
  - Formal parameters are well used (typing)
  - Effective parameters conform to formal ones
  - Usages of generated entities are coherent

Parameterization on values

- A value is a simple concept : value + type
- The formal parameter is typed and **hides its potential value**
- One can use it and verify its type
- Effective values can be automatically substituted on all occurrences in **any document**
  - Simple substitution, macrogeneration, cpp, sed
- Checking *intensity depends on available knowledge:*
  - Nothing: macrogeneration
  - Classic typing: static checking at compilation time, or at binding and run times
  - Assertions, formal specifications: dynamic checking, proofs…

Principles of genericity

- A lot of things are generic: *medicines…*
- In programming, genericity is a form of type genericity
- In languages:
  - Imperative (pioneer): *CLU, LPG, Euclide, Ada*
  - OO: *Eiffel, then C++, C#, Java*
- This is always a constant search for tradeoffs between:
  - Flexibility in derivations
  - Checking safety
  - Cost at compilation and run times
  - Simplicity, readability

Generic OO languages

- Eiffel (pioneer)
  - Powerful and complex multiple inheritance
  - Primitive types as expanded objects (no limitations for genericity)
  - No genericity information at runtime (a simple technique of static typing)
  - Genericity very readable and very simple
  - Constrained genericity through abstract classes
  - Automatic constraints with **like** object / **like** current
  - No introspection on effective types

Generic OO languages (2)

- C++
  - Macro-generation at compilation or linking times
  - Checking done through the linkers => error messages are *strange*
  - No constrained genericity (implicit genericity in signatures)
  - No guarantee that the introspection of effective types will be possible at runtime

Generic OO languages (3)

- C#
  - Instantiation of generic types at *runtime*
  - Genericity constrainted by class and interface
  - Type checking by the compiler and possible introspection at runtime
- Java
  - Ten years to put chunks of genericity inside…
  - Compatibility at the code level between generic/non-generic software
  - Very sophisticated, but complex
  - Inconsistencies and performance loss due to compability constraints (cast, type erasure…)
  - Some problems with primitive types and arrays

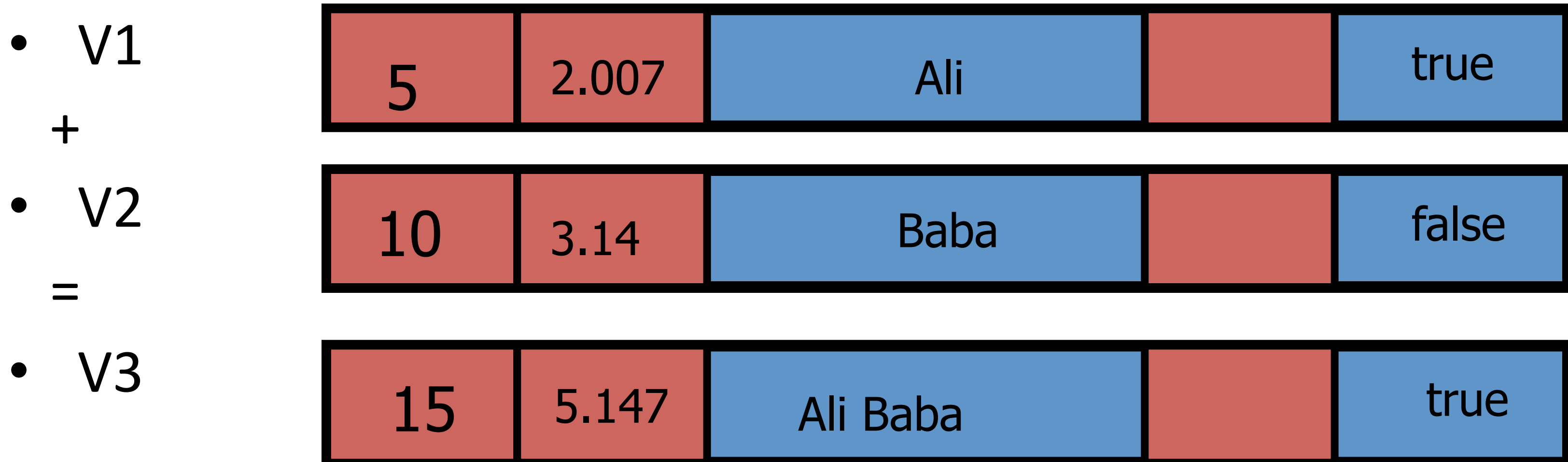
Principles of dynamic genericity

- An object carries information on its properties (through reflexivity) and on its co-instances' properties

- The technique si similar to the **prototype** design pattern
- The formal parameter is an attribute of type T
- All properties of T can be used in the model
- Substitution is dynamic, by a simple (polymorphic) assignment
- Possible checking: by dynamic reflexivity, very flexible but costly…

Example: polymorphic vector

- Addition of two polymorphic vectors:



- Advantages:
  - Get more flexibility than with static typing (with which all elements have the same type)
  - No concession on rigorous typing
  - While doing dynamic checking

Solution #1 : No static checking

- All elements are of type Object
- One checks by introspection that two elements of the same index are compatible for the addition
- The method « plus » of the first element is used to add the second one

- This is possible in all OO language with introspection

Solution #2: mixed checking, static and dynamic

- The necessary and sufficient static typing is added to the previous solution: *All elements are of type Monoid*

```
class PolymorphicVector <Monoid>
        extends ArrayList<Monoid> {
...

}
```

Solution #2: mixed checking, static and dynamic (cont'd)

- One can statically constrain:

  PolymorphicVector <Number>

- One can dynamically constrain:

```
class DynamicMonoVector
        extends ArrayList<Monoid> implements Monoid {
        Monoid prototype ; // fournit le +
        Void DynamicMonoVector(Monoid  type){
            super(); prototype = type;
}
// Utilisation
DynamicMonoVector pvi = new DynamicMonoVector(new Integer(0));
```

Other possibilities

- By combining static and dynamic approaches, one has various possibilities to choose:
  - The degree of polymorphism: all of the same type, same sur-type, same type two by two…
  - When the constraint is defined and checked : at compile or run times

Université Nice SOPHIA ANTIPOLIS

Conclusions on genericity

- Genericity allows for factorization and reuse of knowledge and know-how with guaranteed quality
- Can everything be made generic ? NO!
- The reuse of *architectures or canvas* to solve a general problem cannot be, most often, described by a generic unit.
- *Solutions:*
  - Design patterns
  - Frameworks
  - Domain Specific Languages
  - Dedicated systems (DB, IDE…)
  - Software Product Lines